Efficient and Secure ECC Implementation of Curve *P-256*

Mehmet Adalier
Antara Teknik LLC

www.antarateknik.com

This material is based upon work supported by the National Institute of Standards and Technology (NIST) under cooperative agreement 70NANB14H289. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of NIST.

Motivation

Border Gateway Protocol is vulnerable to malicious attacks that target the control plane

- Prefix/sub-prefix hijacks
  - Steers traffic away from legitimate servers
- Prefix squatting
  - Hijacks a not-in-service prefix and sets up spam servers
- AS path modification (Man-in-the Middle) attacks
  - Modifies AS path causing data to flow via the attacker
- Route leaks
  - Announces routes in violation of ISP policy, thereby redirecting traffic via the attacker

The exploitations commonly result in DDoS, spam, misrouting of data traffic, eavesdropping on user data, etc.

Courtesy of Kotikalapudi Sriram

Motivation

- IETF currently developing BGPSEC (BGP with Security) to provide
  - Route Origin Validation
  - Path Validation
- ECDSA *P-256* is being used for BGPSEC AS-path signing and verification
  - "BGPSEC Protocol Specification," Jan 19, 2015,
- ECDSA *P-256*
  - Provides 128-bit security
  - Approved for protecting National Security Systems (Suite B)

The performance efficiency of ECDSA *P-256* is imperative to meet strict Internet routing table convergence requirements

Optimizations and Considerations

- Multi-level Optimizations to maximize performance
  - Algorithmic Optimizations
  - Group Level Optimization
  - Field Level Optimizations

- Considerations
  - Potentially millions of Public Keys necessitate innovative data handling methods for routines using Public Keys
  - Multi-segment path verifications require thread-safe implementations to maximize system resources
  - Side-channel resiliency required for sign operation

Optimizations must maintain and enhance the security of the implementation under all use-cases

ECDSA Sign and Verify Algorithms

ECDSA Sign

1. Generate k and k^{-1} **P3** **P2**
2. Compute $R = kG$ **P1**
3. Compute $r = x_R \bmod n$ **P4**
4. Compute H = Hash (M)
5. Convert the bit string H to an integer e : where
   $e = \Sigma^{H}_{(i=1)} 2^{H-i} * b_{i,}$
6. $s = (k^{-1} * (e + d * r)) \bmod n$
7. Return (r, s)

ECDSA Verify

1. Is r' and s' in $[1, n - 1]$?
2. Compute H'= Hash (M')
3. Convert the bit string H' to an integer e:' where:
   $e' = \Sigma^{H'}_{(i=1)} 2^{H'-i} * b_{i,}$
4. Compute $w = (s')^{-1} \bmod n$ **P2**
5. $u_1 = (e' * w) \bmod n$
6. $u_2 = (r' * w) \bmod n$
7. $R = (x_R, y_R) = u_1G+u_2Q$ **P1**
8. Compute $v = x_R \bmod n$
9. Compare v and r'. If $v = r'$ output VALID

Px –Priority Optimization Area

Algorithmic Optimizations 1

ECDSA Sign

1. Generate k *and* k^{-1} (P3, P)
2. Compute $R = kG$ (P1)
3. Compute $r = x_R \bmod n$ (P4)
4. Compute H = Hash (M)
5. Convert the bit string H to an integer e : where

   $e = \Sigma^{H}_{(i=1)} 2^{H-i} * b_{i,}$
6. $s = (k^{-1} * (e + d * r)) \bmod n$
7. Return (r, s)

k* and *k*$^{-1}$ *Generation

~15k to 25k cycles

Options:

1. Pre-compute & safely store
2. Asynchronously compute on a different core

Considerations:

- Leaking information on k is detrimental
- For generating k follow NIST Guidelines and Best Practices

Multiplicative Inverse

Options:

1. Fermat's Little Theorem
   - Constant time imp ~18k cycles
2. Half GCD
   - Very fast when variable time

Algorithmic Optimizations 2

ECDSA Sign

1. Generate k *and* k^{-1}
2. Compute $R = kG$
3. Compute $r = x_R \bmod n$
4. Compute H = Hash (M)
5. Convert the bit string H to an integer e : where
   $e = \Sigma^{H}_{(i=1)} 2^{H-i} * b_i,$
6. $s = (k^{-1} * (e + d * r)) \bmod n$
7. Return (r, s)

Observation: The most compute intensive calculations do not have any dependency on the message to be signed

Options:

1. Pre-compute r & safely store
2. Asynchronously compute r on a different core
3. Proprietary methods

Considerations:

- Secure implementations are not trivial

Substantially reduces sign operation latency

Group Level Optimizations 1

INPUT: $k = (k_{t-1}, \ldots, k_1, k_0)_2$, P $\in E(F_q)$

OUTPUT: Q = kP

1. Q← O
2. For i from 0 to $t-1$ do
   2.1 If $k_i = 1$ then Q←Q + P
   2.2 P←2P
3. Return (Q)

Right to Left Binary Method for Point Multiplication

Evaluation time: 0.5mA + mD

P-256 Eval. time: **128A + 256D**

Not SCA resistant

Pt. addition in mixed Jacobian-Affine

$(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : 1)$

$A = X_2 . Z_1^2$, $B = Y_2.Z_1^3$,

$C = A - X_1$, $D = B - Y_1$,

$X_3 = D^2 - (C^3 + 2X_1.C^2)$;

$Y_3 = D. (X_1.C^2 - X_3) - Y_1.C^3$;

$Z_3 = Z_1.C$

Pt. doubling in mixed Jacobian-Affine

$(X_3 : Y_3 : Z_3) = 2(X_1 : Y_1 : Z_1)$, where

$A = 4X_1.Y_1^2$, $B = 8Y_1^4$

$C = 3(X_1 - Z_1^2) . (X_1 + Z_1^2)$, $D = -2A + C^2$,

$X_3 = D$;

$Y_3 = C. (A - D) - B$;

$Z_3 = 2Y_1 . Z_{1,}$

Group Level Optimizations 2

Pre-Calculation:

Take $(K_{d-1}, \ldots, K_1, K_0)_{2^w}$ as the base 2^w representation of k,

where $d = \lceil (m/w) \rceil$, then

$kP = \Sigma^{d-1}_{(i=0)} K_i(2^{wi} P)$

For each i from 0 to $d-1$,

pre-calculate j number of points, where

$j = (2^{w+1}-2)/3$ if w is even;

$j = (2^{w+1}-1)/3$ if w is odd

Storage per Point:

~40KB for P (X, Y)

~60KB for P (X, Y, -Y)

Evaluation time: d(A)

P-256 Eval. time: **~64A**

Evaluation:

INPUT: NAF(k), d, pT (Pointer to pre-computed data table)

OUTPUT: A = kP.

1. Evaluation: A←O
2. For i from 0 to $d-1$ do

   2.1 SafeSelect (Pi),

   use Ki=j to choose the appropriate $P[i][j]$ from Ptable (handle –j)

   2.2 A←A + Pi
3. Return(A)

SCA Resistant Fast Fixed-base NAF Windowing Method for Point Multiplication

Use Chudnovsky + Affine -> Chudnovsky 8M, 3S

Field Level Optimizations 1

- radix-2^{64} representation is quite efficient on a 64-bit architecture compute unit
  - Each field element is unsigned 64-bit type
  - 32-byte values represented with a 4-field element structure
  - Enables effective use of 64-bit CPU instructions
- Special forms of often used parameters enable low-level optimizations
  - p_{256} is a Generalized Mersenne Prime
  - p_{256} = `115792089210356248762697446949407573530` `086143415290314195533631308867097853951`
  - p_{256} = `0xffffffff00000001 0x0000000000000000` `0x00000000ffffffff 0xffffffffffffffff`

Field Level Optimizations 2

- Multi-precision regular/constant time add and subtract modulo prime ops are best implemented in x86-assembly
  - Any Carry or Borrow is easily detected
  - Handled by instructions such as “adcq” and “sbbq”
- Optimized multi-precision multiply and square operations are a must for high performance

Traditional 64-bit multiply in x86

```
mov OP, [pB+8*0]       adc R0, 0
mov rax, [pA+8*0]      add R1, TMP
mul OP                 adc R0, 0
add R0, rax            mov rax, [pA+8*2]
adc rdx, 0             mul OP
mov TMP, rdx           mov TMP, rdx
mov pDst, R0           add R2, rax
mov rax, [pA+8*1]      adc TMP, 0
mul OP                 add R2, R0
mov R0, rdx            adc TMP, 0
add R1, rax            ...
```

64-bit multiply with Broadwell Inst.

```
xor rax, rax                 mulx T1, R’1, [pA+8*2]
mov rdx, [pB+8*0]            adox R’1, R2
                             adcx R3, T1
mulx T1, T2, [pA+8*0]        ...
adox R0, T2
adcx R1, T1
mov pDst, R0

mulx T1, R’0, [pA+8*1]
adox R’0, R1
adcx R2, T1
```

Field Level Optimizations 3

Imperative to optimize reductions

Barrett Reduction modulo p

INPUT: p, $b \geq 3$, $k = \lfloor \log_b p+1 \rfloor$,

$0 \leq a < b^{2k}$, and $\mu = \lfloor b^{2k}/p \rfloor$

OUTPUT: $r = a \bmod p$

1. $q \leftarrow \lfloor a / b^{k-1} \rfloor \cdot \mu$
2. $q' \leftarrow \lfloor q / b^{k+1} \rfloor$
3. $r \leftarrow (a \bmod b^{k+1}) - (q' . p \bmod b^{k+1})$
4. If $r < 0$ then $r \leftarrow r + b^{k+1}$
5. While $r \geq p$ do: $r \leftarrow r - p$
6. Return (r)

Mul+Barrett Red p Cycles ~ 322

Montgomery W-by-W mod p

INPUT: $p < 2^l$ $0 \leq a, b < p$, $l = s.k$

OUTPUT: $r = a.b.2^{-l} \bmod p$

1. $t = a.b$
2. for i 1 *to* k do
   - 2.1 $t_1 = t \bmod 2s$
   - 2.2 $t_2 = t_1 . p$
   - 2.3 $t_3 = (t + t_1)$
   - 2.4 $t = t_3 / 2s$
3. if $t \geq p$ then $r = t - p$
4. else $r = t$
5. Return (r)

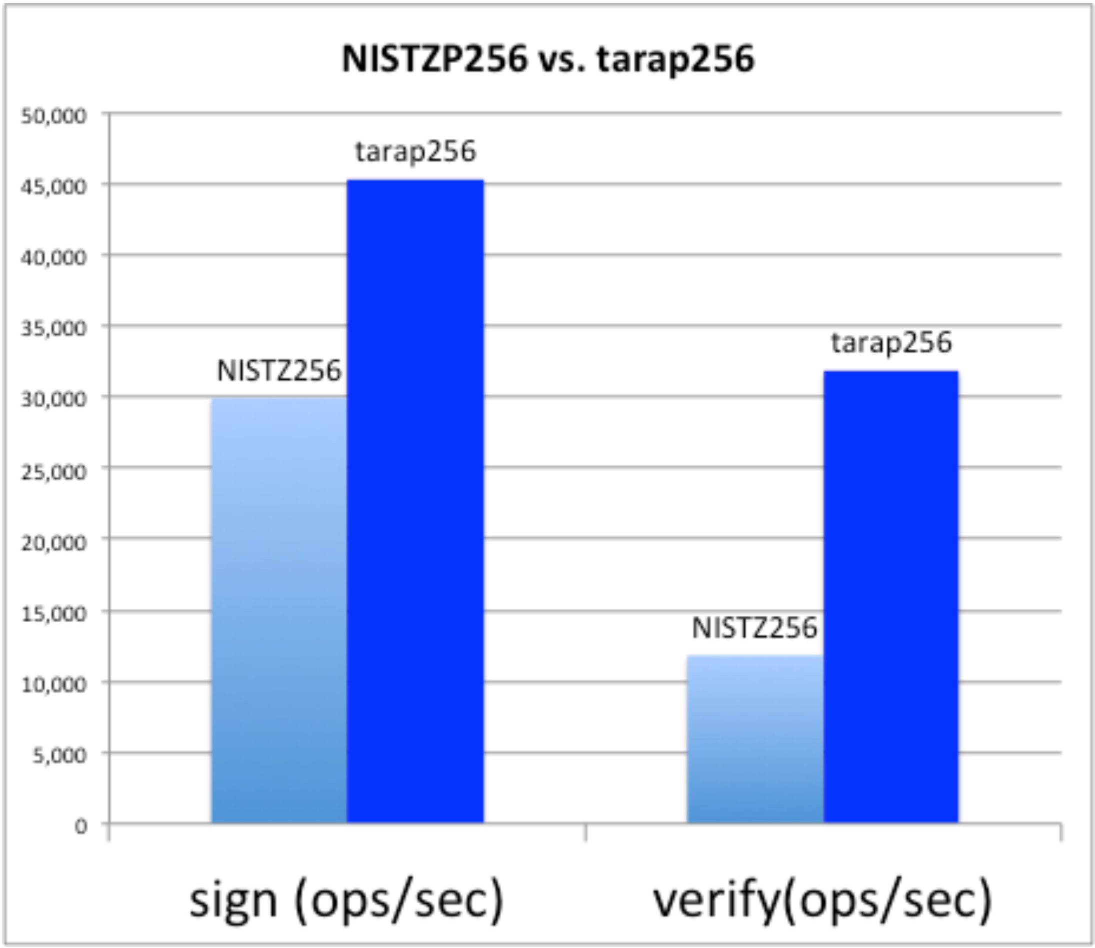
Mul+Mont Red Cycles ~ 298

Results

ECDSA – NISTZ256 vs. *tarap256*
Measured with OpenSSL speed

	ECDSA P-256	
	OpenSSL Speed (NISTZ256)	OpenSSL Speed (tarap256)
sign (ops/sec)	29,938 (1X)	45,300 (1.51X)
verify (ops/sec)	11,842 (1X)	31,805 (2.69X)

(k, k^{-1}) pre-calc'ed (tarap256f):
Sign Perf is 63,807 ops/sec



Measured on Intel® Xeon® E3 1275v3 Single core, Turbo & HT Off

Conclusions

- Performance results indicate that it is possible to implement high performance and secure ECDSA *P-256*

- Our *P-256* implementation, *tara*EcCRYPT™
  - Provides 128-bit security
  - Runs on low-power, low-cost, commercially available CPUs
  - Dynamically supports latest high efficiency CPU instructions
  - Natively thread-safe for multi-CPU and multi-core parallelization
  - <u>Will satisfy BGPSEC Converge Time Requirement</u>